SNS COLLEGE OF ENGINEERING

Kurumbapalayam (Po), Coimbatore – 641 107

An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A' Grade

Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

COURSE NAME : 23ITT101- PROBLEM SOLVING & C PROGRAMMING

I YEAR /I SEMESTER

Unit II – ARRAYS AND STRINGS

Topic : Sorting

Sorting

- Sorting is the process of arranging elements of a collection (such as an array or list) in a particular order, typically in ascending or descending order.
- Sorting is a fundamental operation in computer science and is used in various applications such as searching, data compression, and optimizing algorithms.

Common Sorting Techniques

- Bubble Sort: Simple but inefficient for large datasets. It compares adjacent elements and swaps them if necessary, pushing the largest elements to the end. Not efficient for large datasets due to $O(n^2)$ time complexity.
- Selection Sort: Similar to Bubble Sort in time complexity. It repeatedly finds the minimum element in the unsorted part and places it in the correct position. It is also inefficient with $O(n^2)$ time complexity.

Common Sorting Techniques

- Insertion Sort: Insertion Sort is much better for small arrays or nearly sorted arrays. It works by building the sorted portion of the array one element at a time.
- Merge Sort: A divide-and-conquer algorithm that divides the array into halves, recursively sorts them, and then merges them back. It performs consistently well with O(n log n) time complexity, but it requires additional space.

Common Sorting Techniques

- Quick Sort: Another divide-and-conquer algorithm that works by selecting a "pivot" element and partitioning the array into two parts: one less than the pivot and one greater. It is very efficient in practice but can degrade to $O(n^2)$ in the worst case.
- Heap Sort: A comparison-based sorting algorithm that builds a max-heap (or min-heap) and repeatedly extracts the largest (or smallest) element to build the sorted array. It is efficient with O(n log n) time complexity and uses constant space.

Common Sorting Techniques

- Radix Sort: A non-comparative sorting algorithm that sorts numbers digit by digit. It works well when sorting integers or strings. The time complexity depends on the number of digits (k) in the largest number and the size of the input (n).
- Counting Sort: Works well for integers or categorical data within a limited range. It counts the occurrences of each element and uses these counts to determine the correct positions of elements. It is stable and very efficient when the range of data (k) is small relative to the number of elements (n).

Common Sorting Techniques

- Bucket Sort: Divides the range of input values into a set of "buckets," sorts each bucket individually, and then combines the results. It is efficient when the input is uniformly distributed.
- Shell Sort: An extension of insertion sort that allows elements to be swapped even if they are far apart. The algorithm gradually reduces the gap between elements being compared.

Example : Bubble sort

```
#include <stdio.h>
void bubbleSort(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n-1; i++) {
        for (j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                // Swap arr[j] and arr[j+1]
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
void printArray(int arr[], int size) {
    int i;
    for (i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

```
int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Original array: \n");
    printArray(arr, n);

    bubbleSort(arr, n);

    printf("Sorted array: \n");
    printArray(arr, n);

    return 0;
}
```

```
Original array:
64 34 25 12 22 11 90
Sorted array:
11 12 22 25 34 64 90
```

Example : Quick sort

```
#include <stdio.h>
// Function to swap two elements in the array
void swap(int arr[], int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
// Partition function to choose a pivot
//and arrange elements
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
// choosing the last element as the pivot
    int i = low - 1;  // index of smaller element
 // Rearrange elements based on comparison with pivot
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
// increment index of smaller element
        swap(arr, i, j);
// swap arr[i] and arr[j]
        }
    }
    swap(arr, i + 1, high);
// swap pivot with the element at index i + 1
    return i + 1;
// return the partitioning index
}
```

```
// QuickSort function
void quickSort(int arr[], int low, int high)
 {
    if (low < high) {
// Find the pivot element that is in the
//correct position
        int pi = partition(arr, low, high);
  // Recursively sort the elements //before
and after the partition
    quickSort(arr, low, pi - 1);
 // Left of pivot
    quickSort(arr, pi + 1, high);
 // Right of pivot
    }
}
// Function to print the array
 void printArray(int arr[], int size)
{
  for (int i = 0; i < size; i++)
 {
     printf("%d ", arr[i]);
  }
  printf("\n");
}
```

```
int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
// Example array
    int n = sizeof(arr) / sizeof(arr[0]);
 // Calculate size of the array
    printf("Original array: \n");
    printArray(arr, n);
    quickSort(arr, 0, n - 1);
 // Perform quicksort
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Original array:
10 7 8 9 1 5
Sorted array:
1 5 7 8 9 10

Example : Selection sort

```
#include <stdio.h>

// Function to swap two elements in the array
void swap(int arr[], int i, int j)
 {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

// Selection Sort function
void selectionSort(int arr[], int n)
 {
// One by one move the boundary of the unsorted subarray
    for (int i = 0; i < n - 1; i++)
  {
// Find the minimum element in the unsorted array
        int minIndex = i;
        for (int j = i + 1; j < n; j++)
      {
            if (arr[j] < arr[minIndex])
          {
                minIndex = j;
            }
        }
```

```
 // Swap the found minimum element
 //with the first element
        swap(arr, minIndex, i);
    }
}

// Function to print the array
void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
  {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

```
int main() {
    int arr[] = {64, 25, 12, 22, 11};
// Example array
    int n = sizeof(arr) / sizeof(arr[0]);
// Calculate size of the array
    printf("Original array: \n");
    printArray(arr, n);
    selectionSort(arr, n);
// Perform selection sort
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

```
Original array:
10 7 8 9 1 5
Sorted array:
1 5 7 8 9 10
```

Example : Remove duplicates from a sorted array

```
#include <stdio.h>

int remove_duplicates(int arr[], int n)
{
    if (n == 0)
   {
        return 0; // Return 0 if the array is empty
    }

    // Initialize the index for the unique elements
    int slow = 0;

    // Iterate through the array starting from the second element
    for (int fast = 1; fast < n; fast++)
   {
        if (arr[fast] != arr[slow])
       {
            slow++;
  // Move slow pointer to the next position
            arr[slow] = arr[fast];
// Update the unique element at slow pointer
        }
    }
// Return the length of the array with unique elements
     return slow + 1;
}
```

```
int main() {
    int arr[] = {1, 1, 2, 2, 3, 3, 4, 4};
    int n = sizeof(arr) / sizeof(arr[0]);

    // Call the function to remove duplicates
    int new_length = remove_duplicates(arr, n);

    // Print the array after removing duplicates
    printf("Array after removing duplicates: ");
    for (int i = 0; i < new_length; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Array after removing duplicates: 1 2 3 4

Choosing the Right Sorting Algorithm

- For small arrays or nearly sorted data, Insertion Sort or Bubble Sort might work well.
- For large datasets, Merge Sort, Quick Sort, or Heap Sort are more efficient.
- For sorting integers with a limited range, Counting Sort, Radix Sort, or Bucket Sort may be the best options.


Thank
you